

DTIC FILE COPY

Naval Research Laboratory

Arlington, DC 20375-5000



2

NRL Memorandum Report 6768

## Software Interface for the Touch-Sensitive Menu of the Technician's Assister System

JOSEPH A. MOLNAR

*Communication System Branch  
Information Technology Division*

SONIA FALETTI\*

*\*Synertech Associates, Inc.  
1600 Wilson Blvd.,  
Arlington, VA 22209*

December 20, 1990


DTIC  
ELECTE  
DEC 26 1990  
S B D  
Ca

AD-A230 246

REPORT DOCUMENTATION PAGE			Form Approved OMB No 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503</small>				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 1990 December 20	3. REPORT TYPE AND DATES COVERED Final (1989 to present)		
4. TITLE AND SUBTITLE  Software Interface for the Touch-Sensitive Menu of the Technician's Assister System		5. FUNDING NUMBERS  PE - 25620N WU - DN157139		
6. AUTHOR(S)  Joseph A. Molnar and Sonia Faletti*				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)  Naval Research Laboratory Code 5524 Washington, DC 20375-5000		8. PERFORMING ORGANIZATION REPORT NUMBER  NRL Memorandum Report 6768		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)  Naval Oceanographic and Atmospheric Research Laboratory Stennis Space Center, MS 39529-5004		10. SPONSORING/MONITORING AGENCY REPORT NUMBER		
11. SUPPLEMENTARY NOTES  *Synertech Associates, Inc. 1600 Wilson Blvd., Arlington, VA 22209				
12a. DISTRIBUTION/AVAILABILITY STATEMENT  Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words)  A technician's assister expert system was developed using the Fault Isolation System (FIS) shell. The environment for the proposed application is the cramped irregular spaces of a ship's sonar room. A hardware interface for the software system, incorporating a touch screen, was developed and demonstrated. The interface allows touch entry of all commands, using a portable terminal attached directly to the host computer, or indirectly through the auxiliary port of a VT320 terminal. Both single-display and dual-display configurations are available through software control. This interface was developed to replace a standard terminal interface with command menus. Software was constructed to parse the standard command structure for the terminal, and supply the appropriate command structure to provide the touch interface. The software drivers for the touch screen were written in the C language and interfaced to SUN Command LISP through the use of foreign function calls made from the original software interface. The changes to the original FIS software and the drivers for the touch screen are presented. Results indicate a successful implementation. A restructuring of the interface is planned in the final customized software.				
14. SUBJECT TERMS  Expert system Human interface Fault isolation		Touch screen Networking Maintenance		15. NUMBER OF PAGES
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT  UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE  UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT  UNCLASSIFIED	20. LIMITATION OF ABSTRACT  SAR	

## CONTENTS

1.0 INTRODUCTION .....	1
2.0 APPROACH .....	1
3.0 DESCRIPTION OF THE DEECO TOUCH SCREEN .....	2
4.0 SOFTWARE DEVELOPMENT .....	3
4.1 Alterations to FIL Menu .....	3
4.2 Software for Operating the Touch Screen .....	5
5.0 RESULTS .....	6
6.0 CONCLUSIONS AND SUMMARY .....	7
7.0 REFERENCES .....	7
APPENDIX A — Summary and Listing of LISP Functions .....	9
A.1 DETAILED SUMMARY OF FIL SOFTWARE MODIFICATIONS .....	9
A.1.1 FIL Modifications from the File "utility.lisp" .....	9
A.1.2 FIL Modifications from the File "norda.lisp" .....	10
A.2 LISTING OF FIL SOFTWARE MODIFICATIONS .....	11
APPENDIX B — Summary and Listing of C Functions .....	19
B.1 DETAILED SUMMARY OF THE C FUNCTIONS WHICH OPERATE THE TOUCH SCREEN .....	19
B.2 LISTING OF THE C FUNCTIONS WHICH OPERATE THE TOUCH SCREEN .....	21



iii

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

# SOFTWARE INTERFACE FOR THE TOUCH-SENSITIVE MENU OF THE TECHNICIAN'S ASSISTANT SYSTEM

## 1.0 INTRODUCTION

This report presents the details and evaluation of efforts to provide a touch screen interface as an option to the Technician Assistant System/Fault Isolation Layer (TAS/FIL), a version of the more general Fault Isolation System software package[1]. It is intended for users of the software package. Others may find this report useful in establishing dual-display or touch capabilities, for a terminal with more than one bi-directional serial port. For users of the software package, the report provides details of the software used in providing a touch-sensitive interface. It should be used to maintain the software functionality during subsequent modifications. The technique should interest those seeking to add a similar capability to existing software. The areas of general interest describe the various configurations provided with a touch screen. Primarily, two displays were configured independently to display information. Commands were entered with either device. The hardware [2] involved facilitates the concept of independent input/output (I/O).

LISP was the primary language used for the TAS software. The C language was used for the software that added the touch screen capability. Development was performed on a SUN workstation. The expert system, when completed, had acceptable execution performance. Since a goal of the project was to make the TAS accessible to technicians, the concept of adding the portable touch-sensitive interface was conceived. The device is suitable because it weighs less than 10 lbs. and does not require a keyboard. Technicians often perform maintenance in areas that have irregular surfaces and minimal space, so portability issues were important. The disadvantage of this concept is that it requires a serial connection.

The concept addresses the following scenarios for the software implementation. First, the interface must be convenient for a technician performing a maintenance action. The work area will generally be the cramped equipment compartment on a ship. Second, most maintenance actions are initiated at the console of the warfare system, and the technician maintaining a warfare system is often also the operator. Therefore, the TAS system interface should be capable of extending beyond a console environment. Third, the TAS system interface should be suitable for interaction with the software training features. The interface should be suitable for application in a depot maintenance environment, where space is not critical.

The TAS is an expert system to aid a technician in isolating faults within the AN/SQS-53B sonar system. It was developed utilizing the FIS expert system shell. FIS is still in development and subject to change. FIS formed the core of the model based portion of the TAS system. The TAS design incorporated a strategy that left FIS intact, allowing for ease of FIS system upgrades. All new functionality or enhancements were to exist in software functions external to the body of FIS. The standard interface for FIS was a terminal that displayed a menu of selections. The software described integrates the touch screen as an alternative I/O device for TAS control. Infrared sensors in the bezel of the electroluminescent display (DeeCO M3 SealTouch), registered touch activity. Software controls the presentation of active touch-sensitive areas on the touch screen. The hardware registers the selection of commands. Both text and graphics may be displayed on the screen.

## 2.0 APPROACH

Various display configurations were required to meet the implementation strategy. Software presented the user with options to change the hardware configuration at any time using the main demonstration menu. There were three configurations: the terminal connected directly to the computer, the touch screen connected directly to the computer and the touch screen connected through the VT320 terminal to the computer. The first configuration would apply to stationary scenarios such as training. The maintenance scenario required portability. The remaining two configurations provide the necessary portability. The last configuration is achieved by using the VT320

terminal in the first configuration and attaching the touch screen to the VT320 printer port. This allows information to be sent to both displays independently. In this configuration the input capabilities of both devices remain functional, e.g., keyboard or touch entry of commands. This configuration was intended for a depot maintenance environment or the initial phases of fault isolation that would be encountered in the vicinity of the operator's console.

FIS is continually being upgraded. This constrained the development of the touch screen interface. All of the software written for the interface, therefore, had minimal interactions with the bulk of the FIS software. The software controlled output to the terminal based on the user's hardware configuration selection. Based on this selection, the software parsed the normal output for terminal and added appropriate control commands to display on the touch screen. The software module that parsed the output and controlled display was written in C. These functions were invoked from LISP as foreign functions. C was chosen because the programmer was more proficient with this language and could write and debug the software more quickly. The software function for interface control examined whether a FIL command menu was being sent to the display (see Fig. 1). The software, then, identified the commands and created touch buttons for each. The software organizes the buttons onto pages. Each page corresponded to a different column of menu commands, as seen in Figs. 2, 3 and 4. Additional buttons were created to change command button pages. Buttons were also used to provide descriptive titles for each set of command buttons. These buttons are not associated with any response command. The most frequently used command column, for each menu list, was determined and displayed as the primary display page. The less frequently used command columns were selected by the user through a touch button. The commands from each menu column were placed on different pages to prevent button crowding; the intent was to reduce miskeying. During the evaluation period, none of the commonly encountered touch screen problems, such as miskeying, were encountered. However, this was reasonable since quite large buttons were used. The object of using the touch screen was not to press the state-of-the-art in touch screen precision, but simply to offer a reliable interface, suitable for work in irregular or confined areas. Because large buttons were used, some commands used in development were removed from the menu. Only the commands essential for troubleshooting maintenance were retained.

### 3.0 DESCRIPTION OF THE DEECO TOUCH SCREEN

The touch screen display can be implemented as a VT100 terminal (e.g., connected directly to a computer), or it can be operated from the printer port of a Digital Equipment Corporation (DEC) VT320 terminal. A separate report [2] describes the hardware interface. Keyboard entry is accomplished in one of three ways: through the touch-sensitive, on-screen keyboard, through an attached IBM PC compatible keyboard, or if connected to a VT320 terminal, through that keyboard. For compactness and portability, the on-screen keyboard was most suitable.

In addition to the standard text capabilities of a VT100 terminal, the touch screen has some special graphics capabilities. Lines, arcs, circles and polygons may be drawn, and polygons may be filled. In this project the graphics capabilities of the touch screen were used to display bar graphs.

The touch sensor provides three options for reporting user touch interactions. A touch-sensitive keyboard was a standard default capability. When activated the keyboard displayed in the lower half of the screen. Touching the lower right corner of the screen acts as a toggle function to activate/deactivate the keyboard. Another option allows software defined, touch-sensitive buttons (e.g., rectangles with labels) to be displayed. When any button is touched, the controller transmits the software definition for the button. As a third option, software could be used to activate a mode that tracks the coordinates of a single touch or trace. The interface software implemented only the first two options.

The touch sensor option facilitates the development of a user friendly interface. Each button was presented as a rectangle surrounding a standard text label. Each button may also possess a response, that is, a message which is sent to either the host computer or the touch screen controller upon button activation. A group of buttons displayed on the screen is called a page. Multiple pages of buttons may be defined, with each page possessing at least one button to allow the user to switch between pages. A button, once defined, may be used on several pages. Button pages are stored in the memory of the infrared controller. There is a limit of 100 pages and 100 buttons.

## 4.0 SOFTWARE DEVELOPMENT

FIL was written in LISP. The standard interface for FIL is a VT100 terminal, with a textual menu presentation. Developing the touch-sensitive interface for FIL required two main steps. First, the original FIL menu was altered to allow recognition of the touch screen. Second, software was written to execute the touch screen control.

### 4.1 Alterations to FIL Menu

A FIL function called "menu" was used to execute menu display. This function displays up to four columns of menu choices, with a heading for each column (See Fig. 1). Each menu option appears under a column heading and in the format "mo: menu-option." The first few characters preceding the colon are an abbreviation of the full name of the menu option. The abbreviation shortens the typing needed when using the original menu with the standard terminal interface. The "menu" function also controls the display format.

<u>INSPECTION</u>	<u>TROUBLESHOOTING</u>	<u>CONTROL</u>
sh: show-history	bt: best-test	co: change-output-device
al: active-list	fa: forget-all	rn: return-to-expert
as: amb-set	fl: forget-last	
sp: show-probs	mt: make-test	
bg: bar-graph	mtn: make-test-num	
Enter command or ? -->		

Fig. 1 - FIL Menu as Displayed on a Terminal

Each menu option displays as a rectangular button on the touch screen. Separate pages display menu options for each of the column headings. Displaying all the menu options on one page would clutter the screen with many small buttons, thus promoting miskeying. The actual size of the button facilitated unambiguous button selection. Each screen displays a heading (the column title), the menu options, and paging buttons to allow viewing other screens, see Figs. 2-4 where each page displays information for one column of the menu.

The value of a global variable indicates the output device selection. This variable was initialized to the value for the standard terminal. The user changes the interface by selecting the appropriate option from the menu. When choosing the DeeCO display as the output device, the user specifies the type of connection, either directly to the host computer or through the printer port of a VT320 terminal.

The revised "menu" function replaces the original "menu" function. The display is dependent upon the output device flag. When the flag was equal to DeeCO, the function categorized text as menu text or plain text. Plain text used the original display format. The function parsed the menu text, then passed the resulting string to the C button functions.

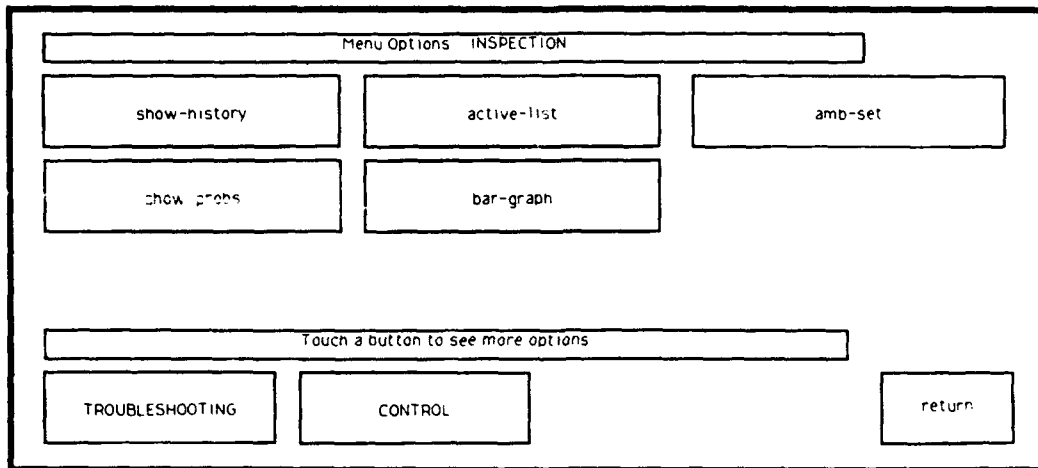


Fig. 2 - Touch Screen Menu--Page 0

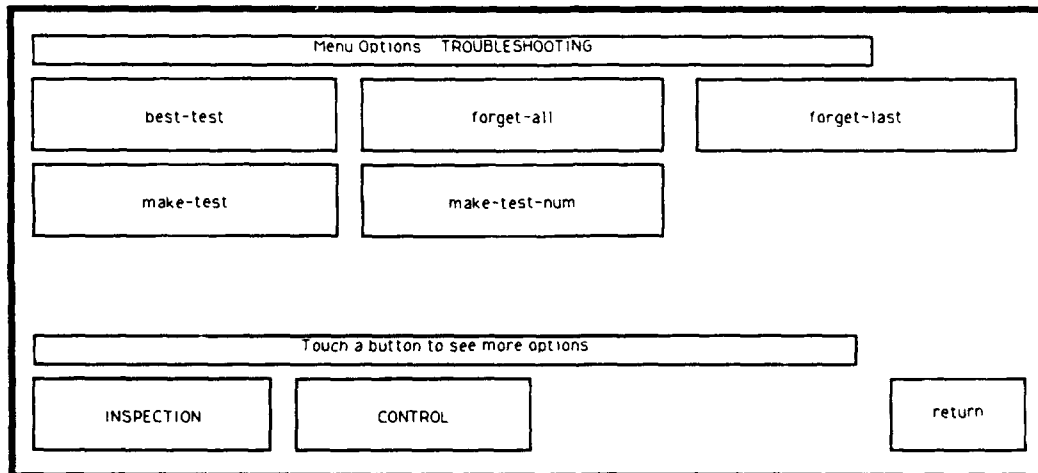


Fig. 3 - Touch Screen Menu--Page 1

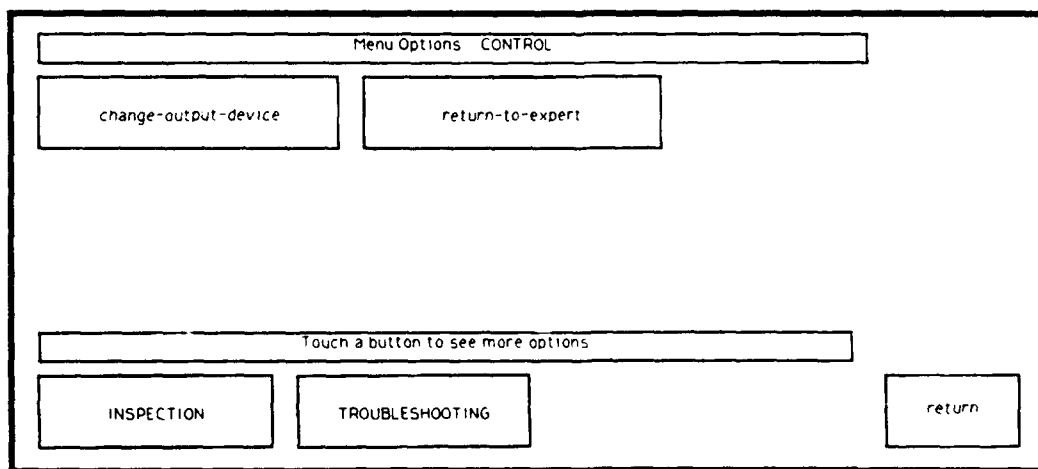


Fig. 4 - Touch Screen Menu--Page 2

## 4.2 Software for Operating the Touch Screen

A package of C functions accepts the parsed FIL menu as a string and produces pages of buttons. The use of C functions required the foreign language interface provided by the Lucid Common LISP compiler. In LISP each menu is in lists, one list for each column. However, in order to pass the menu to a C function, these lists were converted to a single string. Figure 5 indicates the functionality of the software.

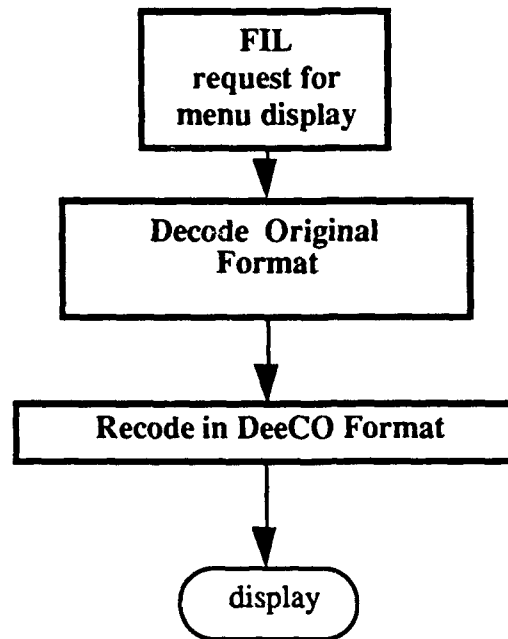


Fig. 5 - Functional flow to achieve touch-sensitive display for TAS on DeeCO display

The C function defines arrays for the column headings, button labels and button responses. It also clears the screen and erases previous buttons from memory. Another function is then called to produce pages of buttons, one page for each column of menu selections. If the menu does not fit the regular format and can not be converted to buttons, the button program stops and returns a signal to FIL. The data in the original format is then displayed.

The commands for operating the touch screen are in ANSI format. Commands in ANSI format begin with a sequence initiator, followed by parameters and then a final character or a terminator. The sequence initiator may be "ESC" (escape) or "CSI" (control sequence initiator). "CSI," equivalent to "ESC [," may only be used in 8 bit mode. "ESC" and "ESC [" may be used in both 7 and 8 bit modes. The parameters are usually numbers separated by semi-colons. Final characters are usually letters, and indicate which command is performed. A terminator, like "ESC \," is used with device-specific commands. Commands in this format are "escape sequences" or "control sequences."

An example of a touch sensor command is the command to turn on the on-screen keyboard: "ESC [ 32 h." (Spaces between the characters are included here for readability, but are not included when sending the control sequence). In this command, "ESC [" is the sequence initiator, "32" refers to the keyboard, and the final character "h," means turn on. The command to turn off the keyboard is "ESC [ 32 l," with the letter "l" meaning "turn off"[3].

There are two steps used to create a touch-sensitive button. The first step is to define the size and position of the button. This is done by specifying the button number and the coordinates for its bottom, left, top, and right sides. The coordinates of a button are defined once and stay the same for all pages on which they appear. However, all of the buttons do not have to appear on all pages. A button will not appear unless it also has a response assigned to it. A button may have different responses on different pages[3]. This software allows the button size to be changed through altering the definitions at the beginning of the program. Changing the button size required



software modification and recompilation. The portion of software which defines the button locations is adaptable to different sizes of buttons. If the buttons will not fit on the screen an error flag is set and the menu is displayed in the original menu format.

Defining the button response is the second step. When a response is assigned to a button several aspects of a button are specified. The label and response may be the same or different. The label is the word or words which appear in the button. The response is the message sent to the host when the button is touched. A button may also have a local response. A local response is a command sent to the touch screen. This was used to switch from one page to another, or for any other touch sensor command. The font size and video state may also be specified. The video state may be either dark buttons (with bright border and text) which become highlighted when touched (the default state) or highlighted buttons which become dark when touched. If a response appeared on multiple pages (e.g., "help" button), it was duplicated with the "copy" command. This command acts as a pointer, and decreases the memory required to store the button information by referring to the first instance of the response instead of restoring on multiple occasions[3].

If only one page of buttons is defined, there is no need to specify a page number. The default page is "page zero." However if multiple pages are desired, it is necessary to indicate the page number before defining the button responses for that page. That page remains the "definition page" until a new page is defined. While the buttons are being defined during the initialization process, either the definition page or another page is displayed. The page being displayed is the "current page." The command "select current page" chooses the page display and also sets that page as the definition page. Buttons defined after that command are immediately displayed. If several pages are defined with this command, each page is shown briefly as the buttons are defined. This means that the user sees all of the pages flash on the screen of the touch screen. The command "select current and definition page" has two parameters: one for selecting the current page and another for selecting the definition page. Buttons defined after that command are not displayed until their page becomes the current page. This technique allows the display to show one page (like an introductory page, the first page, or a blank page) while all the other page definitions are being sent to the infrared controller board[3].

A button is chosen by placing one's finger (or a narrow object like the eraser end of a pencil) within the boundaries of the button. The screen must be touched at only one place at a time. If the screen is touched in several places simultaneously, or if the finger is not entirely within the boundaries of the button, there is no response. It is possible, but difficult, to block the beams without actually touching the screen, as the beams are close to the screen. By using the default button modes the sensor records the coordinates of where the beams are first blocked, so sliding the finger into the button will not work. When a button is properly chosen, it highlights the button. The touch screen beeps and the response is sent. Each of these button modes can be changed - the sensor can register the point where the finger exits the screen instead of where it enters, and the buttons can be set so that they do not highlight or beep[3].

The SealTouch touch screen lacks the capability to add text to a page of buttons. Headings and instructions must therefore be displayed in buttons. In this application, headings were defined as buttons with no response. They were also highlighted, to distinguish them from regular buttons.

The software used several commands to initialize the touch screen before sending pages of buttons to replace the command menu. One command erased all buttons, responses and labels, from the touch screen memory and from the screen. Another command cleared the screen. The on-screen keyboard was switched off to prevent it from interfering with the display of buttons. Also, the mode "clear page on page change mode" was set. This ensures that the current page was erased from the screen before displaying the new page.

## 5.0 RESULTS

The result of this effort was that a touch screen was integrated to provide a touch-sensitive interface for a technician to use the TAS software. The touch screen was demonstrated as a stand-alone terminal connected directly to the host computer, and as an additional display connected via the printer port of the VT320 terminal. The stand-alone configuration performed all of the normal display functions of the terminal, but added the capability of providing touch command entry. In the dual-display configuration, information was displayed to both screens independently. This was particularly useful when presenting both text and graphics.

When using FIL, the first menu currently displays as text. The user may then select the DeeCO touch screen as the output device. Subsequently, all menus display as pages of buttons. The user makes selections by touching buttons. If the selection calls for a new menu, it also displays the commands as buttons. If the selection requires text or graphic display, the display appears as it would on a regular terminal. The menu buttons are large enough to touch easily, and are clearly defined.

Several improvements are suggested for the final implementation of the touch screen. For example, it would be more convenient if the user could designate the touch screen as the output device before the first menu displays. This would be done either by default or by polling the terminal's capabilities. Also, there are currently several points at which the user needs to type a "return." This requires either an auxiliary keyboard, or turning on the on-screen keyboard. A "return" should be included as a mandatory button function on all pages. There are several points in the program where the user is offered several options without calling the menu function. This requires typing one or two characters. The program could be adapted to create buttons for these options as well. The touch screen interface was considered as an alternative output device for the project. The menu data structure was maintained and parsed for display as touch-sensitive buttons. Future efforts should restructure the I/O to address the availability of other interfaces for user interaction.

## 6.0 CONCLUSIONS AND SUMMARY

It was demonstrated that a touch-sensitive menu could be added to FIL. This created an interface that meets the various requirements of operational maintenance and training scenarios. The touch screen interface is portable and suited for operation in irregular work environments. When incorporated with the VT320 terminal, the software could configure the terminals to display independently. This permitted more information to display on an increased display area. Redundancy was provided by having the VT320 keyboard and the touch screen auxiliary keyboard available for command entry, as well as the touch keyboard and touch defined interface. Such redundancy would be unnecessary in real applications. However, it allows two technicians working in separated areas, such as the console and a equipment rack, to have access to the output display and control functions. One obvious difficulty would be inadvertent command entry.

Since the actual commands required to drive the software are short, generally two character commands, the amount of typing saved by this method is small. The advantage resided solely in its portability. The terminal was light weight and the touch capability allows users the limited selection of single touch commands. The focus of adding the touch display was to minimize the interaction and to make those interactions simple. As such, the touch screen and the TAS software function on a level more suitable for troubleshooting. It becomes a tool much like a voltmeter or oscilloscope. As a tool, it can be integrated into the act of maintenance.

## 7.0 REFERENCES

1. F. Pipitone, K.A. DeJong, and W. Spears, "An Artificial Intelligence Approach to Analog Systems Diagnosis," NRL Report 9219.
2. S.G. Faletti and J.A. Molnar, "The Hardware Interface for a DeeCO Touch Screen Terminal in a Technician Maintenance Application," NRL Memorandum Report 6659, May, 1990.
3. Digital Electronics Corporation, *M3 Module Applications Manual*, (Digital Electronics Corporation, Hayward, CA, 1989)



## APPENDIX A

### SUMMARY AND LISTING OF LISP FUNCTIONS

#### A.1 DETAILED SUMMARY OF FIL SOFTWARE MODIFICATIONS

In this section each LISP function altered or added to FIL is explained. The first set of functions is in the file "utility.lisp." The second set of functions is in the file "norda.lisp." All functions listed are either new functions, modifications to existing functions, or functions which were unaltered but renamed for integration purposes. The function names appear italicized between quotation marks. They are explained sequentially, although some of the functions are nested within other functions.

##### A.1.1 FIL Modifications from the File "utility.lisp"

The general flow and interconnection relationships of the functions are shown in Fig. A1. A description of the functions and their relationship follows:

*'menu'* - (Appendix A.2, - lines 1-23) This LISP function is called by FIL any time menu options or information are to be displayed. It receives as input the lists (up to a maximum of four) to display. It uses the global variable "pportp" to determine whether the information should display on a terminal or on the touch screen. It also tests whether the information is a simple string, since all menus are simple strings. If the information should display on a terminal, or if it is not a menu, *'menu'* passes the information to *'vmenu'* to display. If the information should display on the touch screen, *'menu'* calls *'menu-string'*. This converts the list or lists to one string. The C function *'button'* then produces buttons for the touch screen. *'button'* returns true if it successfully produces button commands. If *'button'* is unsuccessful, it returns false and *'vmenu'* is called.

*'vmenu'* - (Appendix A.2, - lines 24-56) This is FIL's original menu function, with a new name. It is called by *'menu'* if the information should display on a terminal or if the information is not a set of menu options. It accepts up to four lists of information and displays them in columns, with a heading for each column and a line of dashes below each heading.

*'menu-string'* - (Appendix A.2, - lines 57-118) This LISP function accepts up to four lists from *'menu'* and converts them to one single string. The string is in the same format as the display produced by *'vmenu'*. It uses the functions *'initstr'*, *'addstr'*, *'tabstr'*, and *'endline'* to set up the string. The string is contained in a global variable called *'\*menu-string\*'*.

*'initstr'* - (Appendix A.2, - lines 120-126) This LISP function initializes the variable *'\*menu-string\*'*.

*'addstr'* - (Appendix A.2, - lines 128-132) Given a string, this LISP function appends it to *'\*menu-string\*'*. It also advances the counter (*'\*string-length\*'*) which holds the length of the current line of the string.

*'tabstr'* - (Appendix A.2, - lines 134-137) Given a number indicating a position on a line, this LISP function fills the *'\*menu-string\*'* with spaces from the current position up to the position indicated by the input number. This serves as a tab function and places the menu options in columns.

*'endline'* - (Appendix A.2, - lines 139-142) This places a carriage return at the end of each line of *'\*menu-string\*'*.

### A.1.2 FIL Modifications from the File "norda.lisp"

The general flow and interconnection relationships of the functions are shown in Fig. A2. A description of the functions and their relationship follows:

"norda-demo-interface" and "norda-display-demo-options" - (Appendix A.2, - lines 143-201) These functions display the options for the main demonstration menu and implement the options chosen. "Change-output-device" was added to this menu to allow the user to change the current output device. Choosing the menu option "change-output-device" causes the function "norda-change-device" to execute.

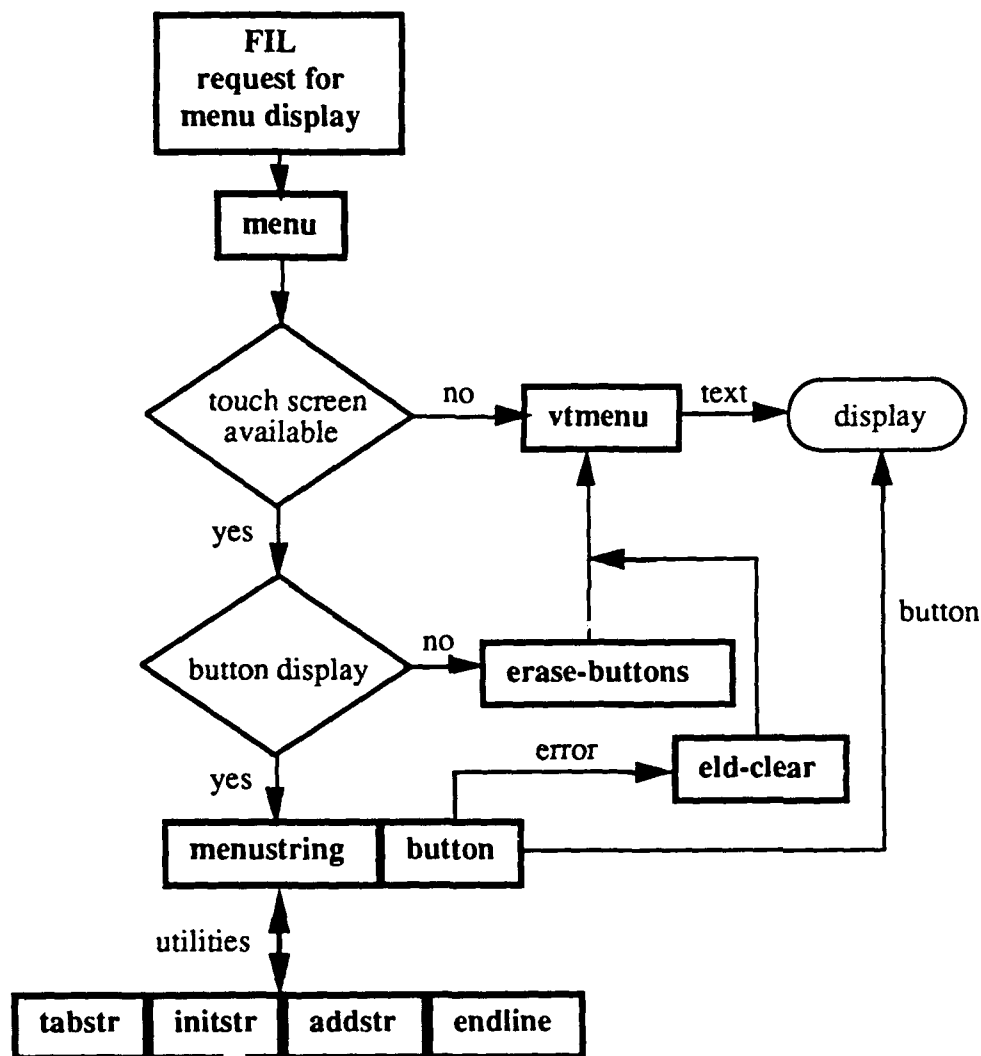


Fig. A1 - General flow relationship of enhanced display functions for FIL output

"norda-change-device" - (Appendix A.2, - lines 258-275) This function produces a secondary menu to allow the user to select a new output device. It uses the function "display-device-options" to display the menu. The options are:

1. A regular terminal (the default option)
2. The DeeCO touch screen connected to the printer port of a VT320 terminal
3. The DeeCO touch screen connected directly to the host computer

If one of the three options is chosen, it calls the appropriate function to change the settings for that option. The three functions are: "*norda-cmd-term*," "*norda-cmd-vt*," and "*norda-cmd-dd*."

"*display-device-options*" - (Appendix A.2, - lines 216-280) This function calls the function "*menu*" to display the three device options and a "quit" option.

"*norda-cmd-term*" - (Appendix A.2, - lines 281-284) This function calls the function "*eld-end*" to direct the output to the terminal and sets the variable "pportp" to 0. The command "finish-output" is used to ensure that the terminal command is sent before proceeding to other commands.

"*norda-cmd-vt*" - (Appendix A.2, - lines 285-288) This function calls the function "*eld-start*" to direct the output to the printer port of the VT320 terminal and sets the variable "pportp" to 1.

"*norda-cmd-dd*" - (Appendix A.2, - lines 289-293) This function calls the function "*eld-end*" to direct the output to the terminal (which in this case is the touch screen) and sets the variable "pportp" to 2.

The following functions ("*eld-start*," "*eld-end*," and "*eld-clear*") all use the function "*print-seq*" to print a device command that includes an "ESC" character. "*Eld-start*" and "*eld-end*" also use the function "*print-seq2*" to print a device command that includes both an "ESC" character and a double quote. "*Eld-start*" and "*eld-end*" are also used by the functions that produce a bar graph on the touch screen.

"*eld-start*" - (Appendix A.2, - lines 224-232) This function sets the VT320 modes needed to operate the touch screen from the printer port of the VT320. The order of the commands is important. First, the terminal is set to VT300 mode. This allows it to be next set to "printer to host mode." "Printer to host mode" enables the VT320 to receive data (e.g., menu choices) from the touch screen. The third command puts the VT320 in "printer controller" mode. In this mode all information from the host is sent to the printer port of the VT320, and not displayed on the screen of the terminal. The last command, "finish-output," ensures that all the preceding commands execute before continuing.

"*eld-end*" - (Appendix A.2, - lines 233-237) This function sends a command to the VT320 terminal to turn off both the "printer controller" and "printer to host" modes. This causes information, from the host, to display on the terminal screen. Then, the terminal is reset to VT100 mode.

"*eld-clear*" - (Appendix A.2, - lines 239-242) This function clears the display of the touch screen.

"*erase-buttons*" - (Appendix A.2, - lines 243-249) This function sends a command to the touch screen to erase all pages of buttons from the memory and display of the touch screen.

"*print-seq*" - (Appendix A.2, - lines 251-254) Given a string that is a device command, this function replaces the tilde (~) with "ESC" and prints the string.

"*print-seq2*" - (Appendix A.2, - lines 255-257) Given a string that is a device command, this function replaces the "Q" with a double quote, tilde with "ESC," and prints the string.

"*norda-definecfunctions*" - (Appendix A.2, - lines 202-222) This function executes automatically when FIL is invoked. It defines and loads C functions used by FIL. Two lines were added to this function. They define and load the C function "*button*." A line was added to set the variable "pportp" equal to zero. This means that FIL expects the output device to be a terminal until another output device is selected

## A.2 Listing of FIL Software Modifications

1.       ::: This menu function allows the use of the touch screen. If the output is
2.       ::: going to a terminal, it calls "vmenu" (the original menu function).
3.       ::: If the output is going to a DeeCo touch screen and there are menu options
4.       ::: in the first column (as tested by "simple-string-p"), it calls menu-string
5.       ::: to put the menu in a string, then passes the string to "button," a
6.       ::: C function. If the string contains menu options, "button" sends them
7.       ::: to the touch screen as pages of touch-sensitive buttons. If the string
8.       ::: contains only text, "button" returns a 0 and "vmenu" is called.

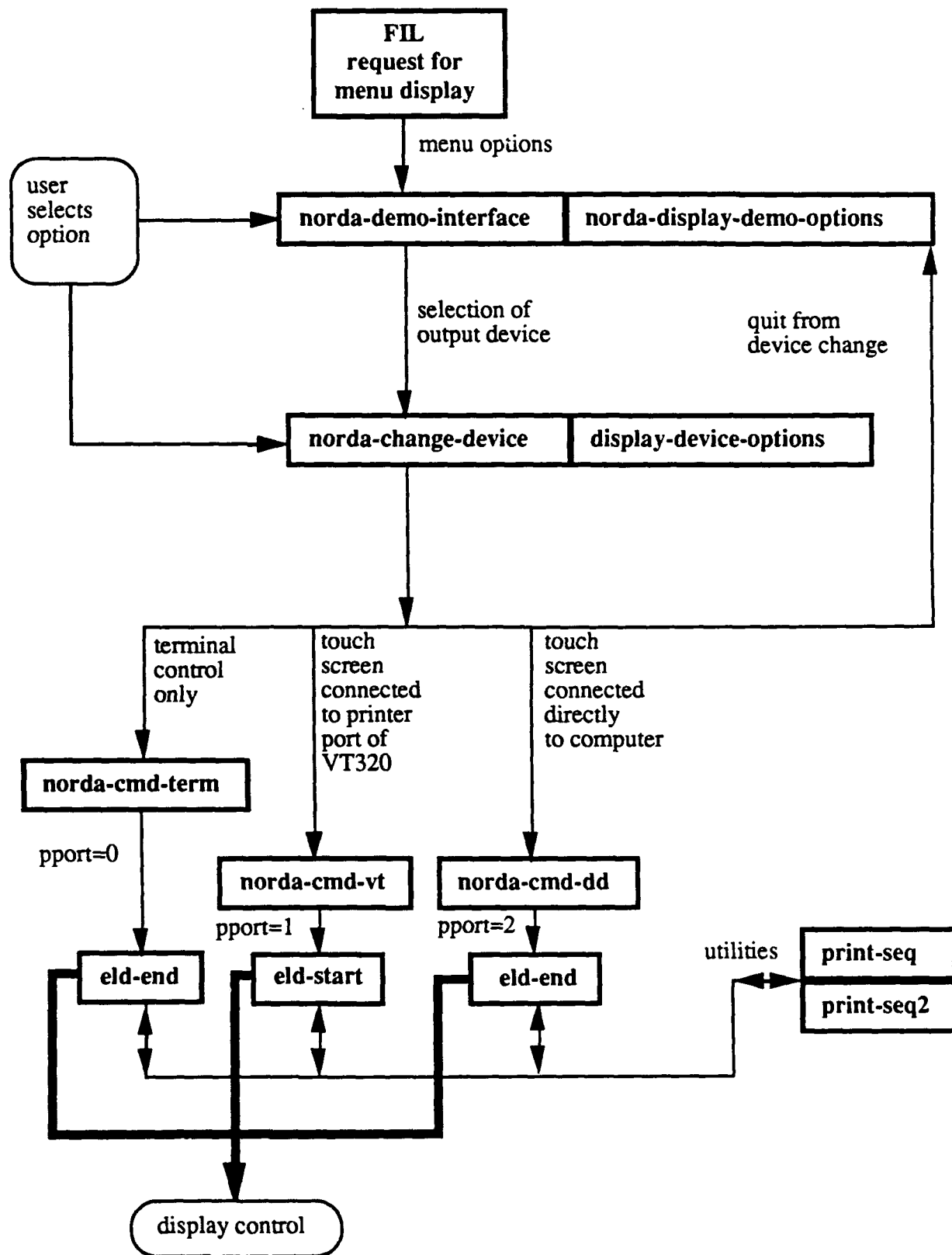


Fig. A2 - FIL functional flow for output device control

```

9. ;; "erase-buttons" and "eld-clear" are defined in norda.lisp.
10. ;; added July 26, 1989, by Sonia Goen Faletti
11. ;; "eld-clear" was replaced with "erase-buttons" in the next to last
12. ;; line. 9008.22 sgf
13. (defun menu (l1 &optional l2 l3 l4)
14. (cond
15. ((= pportp 0) ; output on a VT100 or other terminal
16. (vmenu l1 l2 l3 l4))
17. ((or (= pportp 1) (= pportp 2)) ; output on a DeeCo touch screen
18. (cond
19. ((simple-string-p (cadr l1))
20. (menu-string l1 l2 l3 l4)
21. (cond
22. ((zerop (button *menu-string*)) (erase-buttons) (vmenu l1 l2 l3 l4)
23. (t (erase-buttons) (vmenu l1 l2 l3 l4))))))
24. ;; vmenu is the original menu function. Now it is only used for displays
25. ;; on a terminal or for text displays on the touch screen. sgf, July 26, 1989

26. (defun vmenu (l1 &optional l2 l3 l4)
27. (let ((menu (list l1 l2 l3 l4)))
28. (cons-clear-home) (mark-upper-screen)
29. (terpri)
30. (do ((m menu (cdr m))
31. (col 5 (+ col 20)))
32. ((null m))
33. (when (car m)
34. (tab col)
35. (princ (caar m))))
36. (terpri)
37. (do ((m menu (cdr m))
38. (col 5 (+ col 20)))
39. ((null m))
40. (when (car m)
41. (tab col)
42. (princ (fill (princ-to-string (caar m)) #\~))))
43. (terpri)

44. (do ()
45. ((null (position-if #'identity menu)))
46. (do ((m menu (cdr m))
47. (col 5 (+ col 20)))
48. ((null m))
49. (pop (car m))
50. (do ()
51. ((or (null (car m)) (optionp (caar m))))
52. (pop (car m)))
53. (when (car m)
54. (tab col)
55. (princ (caar m))))
56. (terpri)))

57. ;; This function converts a FIS menu
58. ;; to a string which can be passed to the C function "button."
59. ;; added July 26, 1989, sgf

60. (defun menu-string (l1 l2 l3 l4)
61. (progn()
62. (initstr)
63. (cond (l1 (addstr (princ-to-string (car l1))))
64. (t nil))
65. (cond (l2 (tabstr 20) (addstr (princ-to-string (car l2))))
66. (t nil))
67. (cond (l3 (tabstr 40) (addstr (princ-to-string (car l3))))

```



```

68. (t nil))
69. (cond (l4 (tabstr 60) (addstr (princ-to-string (car l4))))
70. (t nil))
71. (endline)
72. (cond (l1 (addstr (make-string
73. (length (prin1-to-string (car l1)))
74. :initial-element #\~)))
75. (t nil))
76. (cond (l2 (tabstr 20) (addstr (make-string
77. (length (prin1-to-string (car l2)))
78. :initial-element #\~)))
79. (t nil))
80. (cond (l3 (tabstr 40) (addstr (make-string
81. (length (prin1-to-string (car l3)))
82. :initial-element #\~)))
83. (t nil))
84. (cond (l4 (tabstr 60) (addstr (make-string
85. (length (prin1-to-string (car l4)))
86. :initial-element #\~)))
87. (t nil))
88. (endline)
89. (do ((menu1 (cdr l1) (cdr menu1))
90. (menu2 (cdr l2) (cdr menu2))
91. (menu3 (cdr l3) (cdr menu3))
92. (menu4 (cdr l4) (cdr menu4)))
93. ((and (null menu1) (null menu2) (null menu3) (null menu4)) (endline))
94. (do ()
95. ((null menu1) nil)
96. (cond ((optionp (car menu1))
97. (addstr (princ-to-string (car menu1)))
98. (return))
99. (t (setq menu1 (cdr menu1)))))
100. (do ()
101. ((null menu2) nil)
102. (cond ((optionp (car menu2)) (tabstr 20)
103. (addstr (princ-to-string (car menu2)))
104. (return))
105. (t (setq menu2 (cdr menu2)))))
106. (do ()
107. ((null menu3) nil)
108. (cond ((optionp (car menu3)) (tabstr 40)
109. (addstr (princ-to-string (car menu3)))
110. (return))
111. (t (setq menu3 (cdr menu3)))))
112. (do ()
113. ((null menu4) nil)
114. (cond ((optionp (car menu4)) (tabstr 60)
115. (addstr (princ-to-string (car menu4)))
116. (return))
117. (t (setq menu4 (cdr menu4)))))
118. (endline))))

119. ::: The following four functions are called by menu-string.

120. ; initstr is the first function that menu-string calls. It sets up a global
121. ; variable *menu-string* that contains the entire multi-line string. There
122. ; is only one string. It is not a list of strings. If it were, special C
123. ; code would have to be written to understand it.
124. (defun initstr()
125. (progn()
126. (setq *menu-string* "")(setq *string-length* 0)))
127. ;
128. ; addstr duplicates the function of a princ, except puts output in a string.
129. ; it also adds the string length independently, in case another line

```

```

130. ; is printed (newline).
131. (defun addstr(str)(setq *menu-string* (format () "~a~a" *menu-string* str))
132. (setq *string-length* (+ *string-length* (length str))))
133. ;
134. ; tabstr takes as its argument the same value that would be given to a tab
135. ; function. It adds spaces according to where the tab is, and uses addstr to
136. ; do this in order to add to the length.
137. (defun tabstr(num)(addstr (make-string (- num *string-length*) :initial-element #\Space)))
138. ;
139. ; newline replicates a terpri function. This can be done without calling
140. ; addstr because the length is reset to zero.
141. (defun newline()(setq *menu-string* (format () "~a~%" *menu-string*)))
142. (setq *string-length* 0))

```

```

143. (defun norda-demo-interface ()
144. (cons-clear-home)
145. ;; the next function will be the same because it will use
146. ;; norda-display-demo-option
147. (mike-help)
148. ;; Removed following cond - init-probs-and-f-rates done in cmd-load-uut.
149. ;; Creates trouble if one uses demo-interface to start up after an error
150. ;; because the build-property-lists function won't work (assoc lists have
151. ;; been compressed). WMS 8709.29
152. ;; (cond
153. ;; (load-uut-p (forget-all) (init-probs-and-f-rates)))
154. (do ((command)
155. (done nil))
156. (done)
157. (prompt 1)
158. (setq command (read))
159. (cons-clear-line)
160. (case command
161. ((sh show-history) (cmd-show-history))
162. ((al active-list) (cmd-show-active-list))
163. ((sf set-fault) (cmd-set-fault))
164. ;; ((s sim-fault) (cmd-sim-fault))
165. ((bt best-test) (cmd-best-test))
166. ((as amb-set) (cmd-show-amb-set))
167. ((sp show-probs) (cmd-show-probs))
168. ((bg bar-graph) (norda-cmd-bar-graph))
169. ((fa forget-all) (cmd-forget-all))
170. ((fl forget-last) (cmd-forget-last))
171. ((li lisp) (cmd-lisp-command))
172. ((lu load-uut) (cmd-load-uut))
173. ((mt make-test) (cmd-make-test))
174. ((mtn make-test-no) (cmd-make-test-no))
175. ((qt q quit) (setq done (cmd-quit)))
176. ((co change-output-device) (norda-change-device)) ;sgf
177. ((m return-to-expert) (norda-cmd-expert 1)) ;cfs 8910.26 writes valids
178. .dat on user exit
179. ((un unix) (cmd-unix))
180. (?) (mike-help))
181. (t (errmsg "~A: command not found." command)))
182. (princ "Press <return> to continue...")
183. (read-line)
184. (mike-help)
185. ))

```

```

186. (defun norda-display-demo-options ()
187. (menu
188. ;; ("INITIALIZATION" "lu: load-uut") deleted (empty), sgf, July 26, 1989
189. ("INSPECTION" "sh: show-history"

```

```

190.      "al: active-list"      ;"cs: change-setup"
191.      "as: amb-set"
192.      "sp: show-probs"
193.      "bg: bar-graph")
194.  ("TROUBLESHOOTING"
195.   ;;      "s: sim-fault"      "sf: set-fault"
196.      "bt: best-test"      "fa: forget-all"
197.      "fl: forget-last"      "mt: make-test"
198.      "mtn: make-test-num")
199.  ("CONTROL"
200.   "co: change-output-device"      "rn: return-to-expert")
201.  nil))

202.  ;; This function loads the foreign
203.  ;; C files into lisp. the call to
204.  ;; this function is in fis-configure.

205.  (defun norda-definecfunctions()
206.  (progn()
207.   (system:define-c-function nordacalreplace(fn) :result-type :integer)
208.   (system:load-foreign-files '("/usr/aic/norda/demo-revised/src/nordacalreplace.o"))
209.   (system:define-c-function get_uut_string() :result-type :integer)
210.   (system:load-foreign-files '("/usr/aic/norda/demo-revised/src/get_uut_string.o"))
211.   ;; These two lines define and load the C function "modulemaker."
212.   ;; It is an option in the "module-editor" menu.  sgf, July 17, 1989
213.   ;   (system:define-c-function modulemaker () :result-type :procedure)
214.   ;   (system:load-foreign-files '("/usr/aic/norda/demo-revised/src/modulemaker.o"))
215.   ;these were commented out for the lisp version of the function cbb 7/30/89
216.   ;; These lines define and load the C function "button."
217.   ;; It is used to send a menu as buttons to the touch screen.  sgf, July 26, 1989
218.   (system:define-c-function button ((str :string)) :result-type :integer)
219.   (system:load-foreign-files '("/usr/aic/norda/demo-revised/src/Sonia/menuf.o"))
220.   ;; pportp is initialized to 0 (the terminal)
221.   (setq pportp 0)
222.   ))

223.  ;;
224.  ;; These are the new versions of eld-start and eld-end. Besides turning the
225.  ;; printer port on and off, they set (and reset) the appropriate VT320
226.  ;; settings for using the touch screen menu.  sgf, 4/19/90
227.  ;;

228.  (defun eld-start ()
229.  (print-seq2 "~[63Qp")      ;set vt300 mode, 8 bits
230.  (print-seq "~[?9i")      ;set printer to host mode
231.  (print-seq "~[5i")      ;set printer controller mode
232.  (finish-output))

233.  (defun eld-end ()
234.  (print-seq "~[4i")      ;turn off controller mode
235.  (print-seq "~[?8i")      ;turn off printer to host mode
236.  (print-seq2 "~[61Qp")      ;set vt100 mode
237.  (finish-output))

238.  ;
239.  ; eld-clear -- Clears the screen
240.  ;
241.  (defun eld-clear ()
242.  (print-seq "~[2J"))

```

```

243.      ; erase-buttons -- Erases all buttons, responses, and pages on DeeCo.
244.      ; Aug. 3, 1989, sgf
245.      ; altered (inserted first two commands) 9008.22, sgf

246.      (defun erase-buttons ()
247. (print-seq "~[2J")      ;erase screen
248. (print-seq "~[>5D")    ;tells sensor that screen has been cleared
249. (print-seq "~[>B"))    ;erase all pages of buttons

250.      ;
251.      ; print-seq -- prints an escape sequence onto the terminal.
252.      ;
253.      (defun print-seq(str)
254. (write-string (substitute #\esc #\~ str)))

255.      ; print-seq2 -- prints an escape sequence with a double quote  sgf
256.      (defun print-seq2(str)
257. (write-string (substitute #\" #Q (substitute #\esc #\~ str))))

258.      ;; These functions were added to allow the user to choose the DeeCo touch
259.      ;; screen as an output device. This option is added to the demo menu.
260.      ;; sgf, july26.1989

261.      (defun norda-change-device ()
262. (cons-clear-home)
263. (display-device-options)
264. (do ((command)
265. (done nil))
266. (done)
267. (prompt "Enter command or quit -->")
268. (setq command (read))
269. (case command
270. ((tn terminal)      (norda-cmd-term))
271. ((dv deeco-from-vt) (norda-cmd-vt))
272. ((dd deeco-direct)  (norda-cmd-dd))
273. ((q qt quit)        (setq done t))
274. (t                  (errmsg "~A: command not found." command
275. )))

276.      (defun display-device-options ()
277. (menu '("DEVICE-OPTIONS" "tn: terminal" "dv: deeco-from-vt"
278. 278. deeco-direct")
279. '("MISC" "qt: quit")
280. nil nil))

281.      (defun norda-cmd-term ()      ;set terminal as output device
282. (eld-end)
283. (finish-output)
284. (setq pportp 0))      ;pportp 0 is terminal

285.      (defun norda-cmd-vt ()      ;set DeeCo from vt320 as output device
286. (eld-start)
287. (finish-output)
288. (setq pportp 1))      ;pportp 1 is DeeCo from vt320

289.      (defun norda-cmd-dd ()      ;set DeeCo as output device
290. (eld-end)
291. (finish-output)
292. (setq pportp 2))      ;pportp 2 is DeeCo directly connected
293.      ;to mainframe

```



## APPENDIX B

### SUMMARY AND LISTING OF C FUNCTIONS

#### B.1 DETAILED SUMMARY OF THE C FUNCTIONS WHICH OPERATE THE TOUCH SCREEN

In this section each of the C functions added to the FIL package for touch screen control is explained. Functions are again noted by being italicized between quotation marks. The source code for these functions are contained in a single file that has been compiled into an object file. The object file is loaded by a LISP instruction in the FIL function "*norda-definectfunctions*." The functions "*button*" and "*buttonpages*" are the major functions. "*Button*" defines the arrays of information for the menu and calls "*buttonpages*," which produces the pages of buttons. Other functions are utilities used by "*button*" and "*buttonpages*." The flow and interrelationship of functions are shown in Fig. B1.

"*button*" - (Appendix B.2, - lines 1-176) This is the main function of this software file. It is called by the LISP function "*menu*." The input to the function is a string which contains a table of menu options. This function processes the menu information of the string and sorts the information into arrays. The function analyzes the string line by line using the functions "*get\_string*" (to access each line) and "*getword*" (to look at each word in the line). The function "*findcolumns*" sets up an array of numbers based on the locations of the column headings. This array of column numbers is used to define the columns and allocate the menu options to the appropriate column. The column headings are placed in an array called "head," the button labels into an array called "label," and the button responses into an array called "resp." "Label" and "resp" are effectively two-dimensional arrays to store the menu options by columns. (Actually, the information is not placed in the arrays, but rather the arrays are arrays of pointers to the labels and responses. Storage space is allocated and pointers assigned by the function "*readin*." ) The second part of this function checks whether the information in the string is a menu. If it is, then the function "*buttonpages*" is called to produce the pages of buttons for the touch screen. If that function returns true (1), the function "*button*" returns true. This indicates that the menu has been successfully converted to a touch screen menu. If "*buttonpages*" returns false (0) or the information in the string is not a menu, then "*button*" returns false. This signifies that a touch screen menu was not produced. This results in the function "*menu*" displaying the information as text, using the LISP function "*vimenu*."

"*buttonpages*" - (Appendix B.2, - lines 177-277) This function is called by "*button*." The input to the function includes the addresses of the heading, response, and label arrays, an array containing the number of menu options in each column, the number of columns, and the maximum number of buttons. The function initializes the touch screen by turning off the on-screen keyboard, erasing all buttons from memory, setting "clear page on page change mode," and clearing the screen. For each page, it produces a label button with the heading for that page, a button for each menu option, a label for the paging buttons, the paging buttons, and a "return" button. The menu buttons are displayed in rows, filling each row, then going to the next row. If they do not fit, an error message is printed and false is returned. The parameters for the dimensions of the buttons are predefined in the package of C functions. They may be altered. A function exists to produce each type of button:

"*makebutton*" - for the menu option buttons

"*makeswitch*" - for the paging buttons

"*makelabel*" - for the labels (called from "*maketoplabel*" and "*makelowerlabel*")

"*Buttonpages*" will return true if all the buttons fit on each page.

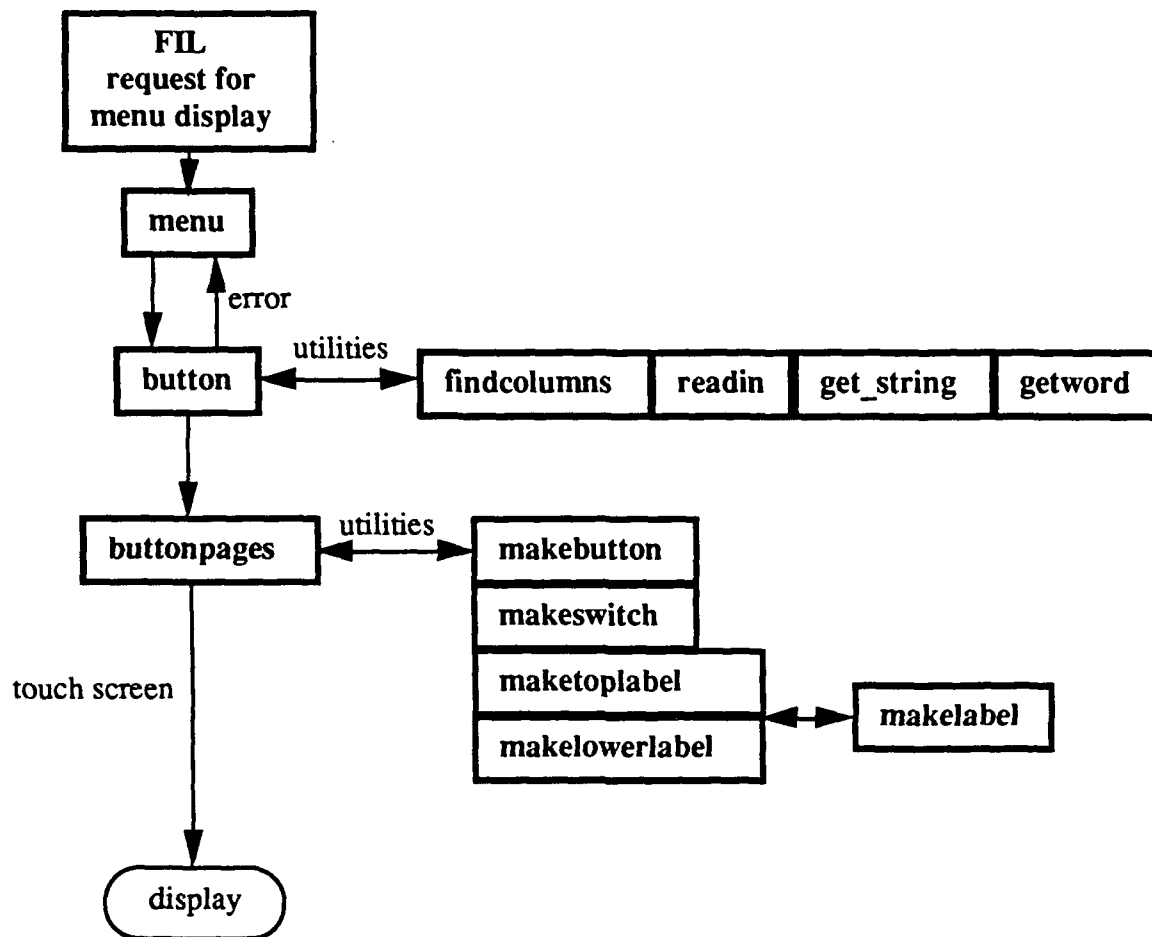


Fig. B1 - Flow diagram of C functions used in display of the button menu

**"readin"** - (Appendix B.2, - lines 359-374) This function is called by **"button"** to assign a pointer to a string. It returns the pointer.

**"get\_string"** - (Appendix B.2, - lines 278-301) This function is called by **"button"** to get one line of the menu string. It uses the pointer **"menu\_string,"** which is global and points to the current position in **"string\_block"** (containing the menu string). It finds the next line (with the end marked by a carriage return), copies it into a new, null-terminated string, and advances the pointer. It returns the new string, or else a null string if the end of the menu string was reached.

**"getword"** - (Appendix B.2, - lines 395-407) This function is called by **"button"** to get the next word from the current line of the menu. A word is defined here to mean a set of characters with no spaces, tabs, returns, or null characters. The input for **"getword"** is a pointer to the next character in the string and a pointer to the location where it will put the word it obtains from the string. After copying the word from the string, it advances the pointer to the first character in the next word and returns the new pointer.

**"findcolumns"** - (Appendix B.2, - lines 302-330) This function is called by **"button"** to set up an array of numbers corresponding to the columns of the menu. The input to the function is the string containing the headings and the array (**"columns"**) to fill with numbers. This array is filled with 1's up to the end of the first heading, 0's until the second heading, 2's where the second heading is, etc. For example:

```

    heading1 heading2 heading3 heading4
1111111111100022222220003333333000444444400

```

It returns the number of columns.

"makebutton" - (Appendix B.2, - lines 331-341) This function is called by "buttonpages" to produce one touch-sensitive button for a menu option. The input to the function includes the button number, response, label and the coordinates for the bottom, left, top, and right sides of the button. The function consists of two print statements - one to define the button's location and the other to define the button's label and response. It does not return a value.

"makeswitch" - (Appendix B.2, - lines 342-350) This function is called by "buttonpages" to produce one paging button. The input to the function includes the button number and label (the heading of the new page), the number of the new page to display, and the coordinates of the button's location. The function consists of two print statements defining the button's location and response. This button is defined to have a local response (changing the page). No response is sent to the host. It does not return a value.

"maketoplabel" - (Appendix B.2, - lines 375-384) This function is called by "buttonpages" to produce the label at the top of the screen that includes the heading for the menu options for its page. The input to the function is the page heading. The function sets up the text and coordinates for this button, then calls "makelabel" to produce the button. It does not return a value.

"makelowerlabel" - (Appendix B.2, - lines 385-394) This function is called by "buttonpages" to produce the label for the paging buttons. The input to the function includes the button number and page number. The function sets up the text ("Touch a button to see more options") and coordinates for this button. If this is the first page (page 0), it calls "makelabel" to produce the button. Otherwise, it copies the lower label from the first page (to save memory). It does not return a value.

"makelabel" - (Appendix B.2, - lines 351-358) This function is called by both "maketoplabel" and "makelowerlabel" to produce explanatory labels. The input to the function includes the button number, label, and coordinates. The function produces a button in reverse video with text, but no response. It does not return a value.

## B.2 LISTING OF THE C FUNCTIONS WHICH OPERATE THE TOUCH SCREEN

```

1. /*****
2. *
3. *      TOUCH SCREEN MENU FOR FIL
4. *
5. *   The purpose of this collection of C functions is to take a FIL menu
6. *   and produce touch-sensitive buttons for it on a DeeCo touch screen. When
7. *   the Lisp function "menu" calls the C function "button," it passes a
8. *   string. This string is in the format in which FIL prints menus on the
9. *   screen of a VT100. The first line has column headings, the second line
10. *   has hyphens under the column headings, and the rest have
11. *   menu options in columns under the headings. "Button" creates one page of
12. *   buttons for each column of selections. The bottom row of buttons on each
13. *   page enables the user to access the other pages of buttons or to return
14. *   to the main menu. Each button for a menu option has a "label" that
15. *   the user sees, and a "response," that the FIL program
16. *   expects.
17. *
18. *   This program was written by Sonia Faletti .
19. *
20. *   --Sonia Faletti, 18 July, 1989
21. *****/

22. #include <stdio.h>
23. #define MAX_LENGTH 500 /* The maximum number of characters in a line */
24. #define TRUE 1
25. #define FALSE 0
26. #define ESC 27 /* ASCII for ESC */
27. #define TILDA 126 /* ASCII for TILDA */
28. #define RETURN 13 /* ASCII for carriage return */
29. #define NUMOPT 5 /* maximum number of columns in menu */
30. #define BLENGTH 19 /* button length */
31. #define BHEIGHT 5 /* button height */
32. #define SLENGTH 15 /* length of switching buttons */

```



```

33.  #define SHEIGHT 5 /*height of switching and help buttons*/
34.  #define RETURNLEN 8 /*length of return button*/
35.  #define LLENGTH 50 /*length of label buttons*/
36.  #define LHEIGHT 2 /*height of label buttons*/
37.  #define HSPACE 2 /*horizontal spacing of buttons*/
38.  #define VSPACE 1 /*vertical spacing of buttons*/
39.  #define BOTTOM 1 /*vertical position of bottom row*/
40.  #define MAXBUTTONS 9 /*maximum number of menu option buttons*/
41.  #define SCREENWIDTH 64
42.  #define SCREENHEIGHT 32

43.  char *menu_string; /* the pointer to string_block */
44.  char string_block[1920]; /* 1920=24*80, a screen full of menu choices */
45.  /* this block is modified by get_string, and */
46.  /* referenced by button. string_block is also */
47.  /* created by button, in the beginning. */

48.  char *get_string(); /* forward declaration of this function */

49.  /*****
50.  *
51.  * button is passed a menu string that is decoded into button commands and
52.  * printed on the DecCo. This function used to be a main program. It is
53.  * the entry point for FIL. The first part of the function reads in
54.  * information from menu_string into arrays. The last part makes
55.  * buttons from the information in the arrays.
56.  *
57.  *****/

58.  int button(passed_menu_string)
59.  char passed_menu_string[];
60.  {
61.  char temp[MAX_LENGTH], temp1[80]; /*for temporary storage of strings*/
62.  char *string; /*pointer for current line*/
63.  int columns[MAX_LENGTH]; /*array: numbers indicate columns*/
64.  char *resp[NUMOPT][MAXBUTTONS]; /*2-dim array of pointers to responses*/
65.  char *label[NUMOPT][MAXBUTTONS]; /*2-dim array of pointers to labels*/
66.  char *info[NUMOPT][100]; /*2-dim array of pointers to info*/
67.  char *head[NUMOPT], *readin(); /*column headings; function declaration*/
68.  int num[NUMOPT]; /*array: number of options for each column*/
69.  int nm, k, k1, c; /*counters*/
70.  int nmcols; /*number of columns*/
71.  int error; /*error flag*/
72.  int col; /*number of current column*/
73.  int buttonmax; /*highest number of menu choices in a column*/
74.  int len; /*length of a string*/
75.  int pos; /*position of ':' in a menu option*/
76.  int commands, text; /*these are flags to determine whether the
77.  menustring contains commands, text, or both*/
78.  /*initializations*/
79.  commands = FALSE;
80.  text = FALSE;
81.  for (k = 0; k < NUMOPT; k++) num[k] = 0;
82.  error = FALSE;
83.  strcpy(string_block, passed_menu_string);
84.  menu_string = string_block;

85.  /* In this section, the column headings are found and stored in an array of
86.  strings. Also, an array of numbers is set up to label the heading
87.  positions */

```

```

88. nncols = findcolumns((string=get_string()), columns);
89. for (k=0; *(string+k) == ' ' || *(string+k) == '\t'; k++);
90. for (c = 0; c < nncols; c++) {
91.     k1 = 0;
92.     while (*(string+k) != ' ' && *(string+k) != '\0') {
93.         temp[k1++] = *(string+k);
94.         k++;
95.     }
96.     temp[k1] = '\0';
97.     head[c] = readin(temp);
98.     while (*(string+k) == ' ') k++;
99. }
100. string=get_string();

101. /* In this section, each line of menu selections is read and the responses and
102. labels are placed in their respective two-dimensional arrays. Commands
103. always have a colon between the response and the label. They may be in one
104. of these formats:
105.     rsp: label
106.     rsp: label or rsp:label
107.     rsp:label
108. Otherwise, the word is assumed to be text (not a menu option). */

109. while(*(string=get_string()) != NULL) {
110.     nm=0; /* nm points to the current position in the string */
111.     do {
112.         while (*(string+nm) == ' ' || *(string+nm) == '\t') nm++;
113.         if (*(string+nm) != '\0') {
114.             col = columns[nm] - 1;
115.             nm += getword(temp, string+nm);
116.             len = strlen(temp);
117.             if (*(temp+len-1) == ':') { /* rsp: label */
118.                 *(temp+len-1) = '\0';
119.                 resp[col][num[col]] = readin(temp);
120.                 nm += getword(temp, string+nm);
121.                 label[col][num[col]] = readin(temp);
122.                 commands = TRUE;
123.             }
124.             else if (*(string+nm+1) == ':') { /* rsp: label */
125.                 resp[col][num[col]] = readin(temp);
126.                 nm += 2;
127.                 nm += getword(temp, string+nm);
128.                 label[col][num[col]] = readin(temp);
129.                 commands = TRUE;
130.             }
131.             else if ((pos=strcspn(temp, ":")) < len) { /* rsp:label */
132.                 strncpy(temp1, temp, pos);
133.                 *(temp1+pos) = '\0';
134.                 resp[col][num[col]] = readin(temp1);
135.                 strcpy(temp1, temp+pos+1);
136.                 label[col][num[col]] = readin(temp1);
137.                 commands = TRUE;
138.             }
139.             else { /* must be text */
140.                 info[col][num[col]] = readin(temp);
141.                 text = TRUE;
142.             }
143.             num[col]++;
144.         }
145.     } while (*(string+nm) != '\0');
146. }

```

```

147.    /* This is to find the maximum number of menu choices in a column. */
148.    buttonmax = num[0];
149.    for (c = 1; c < nmcpls ; c++)
150.        if (num[c] > buttonmax) buttonmax = num[c];

151.    /* If the menustring contains:
152.    only text--make no buttons, return 0
153.    only commands--make buttons, return 1
154.    both text and commands--make no buttons, return 0
155.    (future improvement: put both on the same page)
156.    neither--make no buttons, return 0 */

157.    if (text == TRUE && commands == FALSE) {
158.        printf("%c>B\n", ESC);
159.        fflush(stdout);
160.        printf("%c[2J\n", ESC);
161.        fflush(stdout);
162.        return 0;
163.    }
164.    if (text == FALSE && commands == FALSE) return 0;
165.    if (text == FALSE && commands == TRUE) {
166.        k = buttonpages(head, resp, label, num, nmcpls, buttonmax);
167.        return k;
168.    }

169.    if (text == TRUE && commands == TRUE) {
170.        printf("%c>B\n", ESC);
171.        fflush(stdout);
172.        printf("%c[2J\n", ESC);
173.        fflush(stdout);
174.        return 0;
175.    }
176.    }

177.    /* This function makes pages of buttons for the touch screen. */

178.    buttonpages(head, resp, label, num, nmcpls, buttonmax)

179.    int num[], nmcpls, buttonmax;
180.    char *head[], *resp[][MAXBUTTONS], *label[][MAXBUTTONS];
181.    {
182.    int c, k, k1, row, bottom, left, top, right, oldbutton, oldpage, error;
183.    char temp1[20], temp2[20];

184.    printf("%c>32I", ESC); /* turns off keyboard */
185.    printf("%c[2J", ESC); /* clears screen */
186.    printf("%c>5D", ESC); /* tells sensor that screen has been cleared */
187.    printf("%c>B", ESC); /* erases all buttons, responses, and labels */

188.    /* Each case represents one column (and one page). */

189.    for (c = 0; c < nmcpls ; c++) {
190.        switch (c) {
191.        case 0:
192.            printf("%c>0P\n", ESC);
193.            maketoplabel(head[c]);
194.            fflush(stdout);
195.            break;
196.        case 1:
197.            printf("%c>0;1P", ESC);
198.            maketoplabel(head[c]);
199.            fflush(stdout);
200.            break;

```

```

201. case 2:
202.     printf("%c[>0;2P", ESC);
203.     maketoplabel(head[c]);
204.     fflush(stdout);
205.     break;
206. case 3:
207.     printf("%c[>0;3P", ESC);
208.     maketoplabel(head[c]);
209.     fflush(stdout);
210.     break;
211. default:
212.     return 0;
213. }

214. /* This section creates one button for each menu selection on a page. */

215. for (k = 0, row = 1, k1 = 0; k < num[c]; k++) {
216.     if ((k1+1)*(HSPACE+BLENGTH) > SCREENWIDTH) {
217.         row++; /*If the current row of buttons is full, begin a new one.*/
218.         k1 = 0;
219.     }
220.     bottom = SCREENHEIGHT - LHEIGHT - row * (VSPACE + BHEIGHT);
221.     if (bottom < BOTTOM + SHEIGHT + 2 * VSPACE + LHEIGHT) {
222.         printf("The buttons will not fit on the screen.\n");
223.         return 0;
224.     }
225.     left = HSPACE + k1 * (HSPACE+BLENGTH);
226.     top = SCREENHEIGHT - LHEIGHT - VSPACE - (row-1) * (VSPACE + BHEIGHT);
227.     right = (k1+1) * (HSPACE + BLENGTH);
228.     makebutton(k+1, resp[c][k], label[c][k], bottom, left, top, right);
229.     fflush(stdout);
230.     k1++;
231. }

232. /* This section makes the buttons which allow switching from one page to
233. another. */

234. makelowerlabel(buttonmax + 1, c);
235. fflush(stdout);
236. for (k = 0, k1 = 0; k < nmcols; k++) {
237.     if (k != c) {
238.         bottom = BOTTOM;
239.         left = HSPACE + k1 * (SLENGTH + HSPACE);
240.         top = BOTTOM + SHEIGHT;
241.         right = (k1 + 1) * (HSPACE + SLENGTH);
242.         if (right >= SCREENWIDTH - (RETURNLEN + HSPACE)) {
243.             printf("The switching buttons will not fit.\n");
244.             return 0;
245.         }
246.         if (c == 0 || (c == 1 && k == 0))
247.             makeswitch(buttonmax+k1+2, head[k], k, bottom, left, top, right);
248.         else {
249.             if (k != 0) {
250.                 oldbutton = buttonmax + k + 1;
251.                 oldpage = 0;
252.             }
253.             else {
254.                 oldbutton = buttonmax + 2;
255.                 oldpage = 1;
256.             }
257.             printf("%c[>%d;%d;%dC", ESC, buttonmax+k1+2, oldbutton, oldpage);
258.         }

```

```

259.     fflush(stdout);
260.     k1++;
261. }
262. }

263. /* This section makes the "return" button. */

264. strcpy(temp1, "^M");
265. strcpy(temp2, "return");
266. bottom = BOTTOM;
267. left = SCREENWIDTH - (HSPACE + RETURNLEN);
268. top = BOTTOM + SHEIGHT;
269. right = SCREENWIDTH - HSPACE;
270. if (c == 0)
271.     makebutton(buttonmax+1+nmcols, temp1, temp2, bottom, left, top, right);
272. else
273.     printf("%c[>%d;%d;0C", ESC, buttonmax+1+nmcols, buttonmax+1+nmcols);
274.     fflush(stdout);
275. }
276. return 1;
277. }

278. /******
279. *
280. * get_string: This function returns the next line of text from menu_string.
281. *             It replaced a function which returned input from a file.
282. *             That function's name was gets_1. If there is no more string,
283. *             it returns the null string. The string it returns will be
284. *             null terminated.
285. *
286. *****/

287. char *get_string()
288. {
289. char return_string[MAX_LENGTH];
290. int len;
291. len=strcspn(menu_string, "\n");
292. *(menu_string+len)=NULL;
293. strncpy(return_string, menu_string, len+1);
294. menu_string+=len;
295. if (*(menu_string+1)!=NULL) menu_string++;
296. /* If another string is present, advance to it. Otherwise, keep the menu_string
297. pointer in the same place so that next time a NULL will be returned. The
298. significance of this is that the last character in the string before the
299. null terminator must be a carriage return. */

300. return(return_string);
301. }

302. /* FINDCOLUMNS(String, Columns)

303. This routine receives String, the lowest row in a column header, as input.
304. It computes Columns, an array in which each element contains a number
305. corresponding to the appropriate column that is located there. Columns are
306. given increasing numbers starting at one, and those locations between
307. columns are filled with 0's. For example, the following column header
308. (String) generates Columns as shown:

309. String = " Header1 H2 Header3 Head4 "
310. Columns = "1111111111002200000333333300444440"

```

```

311. Note that leading spaces are filled with 1's. */
312.     findcolumns(string,columns)
313.     char *string;
314.     int columns[];
315.     {
316. int k = 0, num = 0;
317. while (string[k] == ' ') columns[k++] = 1;
318. if (k == 0) {
319.     columns[k++] = 1;
320.     num = 1;
321. }
322. while (string[k] != '\0') {
323.     if (string[k] == ' ') columns[k] = 0;
324.     else if (string[k-1] != ' ') columns[k] = num;
325.     else columns[k] = ++num;
326. };
327. }
328. while (k < 200) columns[k++] = 0;
329. return(num);
330. }

331. /* This function produces one button for the touch screen, with a label and a
332. response. */

333.     makebutton(number, response, label, bottom, left, top, right)
334.     char *response, *label;
335.     int number, bottom, left, top, right;
336.     {
337. /*This section defines the button location.*/
338. printf("%c[>%d;%d;%d;%d;%dB",ESC, number, bottom, left, top, right);
339. /* This section defines the button response. */
340. printf("%cP%c%d;%s^M:%s%c\\", ESC, TILDA, number, response, label, ESC);
341. }
342. /* This function makes a button for switching pages, using the "local response"
343. capability of the touch screen. */

344.     makeswitch(number, heading, newpage, bottom, left, top, right)
345.     char *heading;
346.     int number, newpage, bottom, left, top, right;
347.     {
348. printf("%c[>%d;%d;%d;%d;%dB", ESC, number, bottom, left, top, right);
349. printf("%cP%c%d/\n9B>%dP:%s%c\\", ESC, TILDA, number, newpage, heading, ESC);
350. }

351. /* This function produces one label button with no response. */

352.     makelabel(number, label, bottom, left, top, right)
353.     char *label;
354.     int number, bottom, left, top, right;

```

```

355.     {
356.     printf("%c[>%d;%d;%d;%d;%dB",ESC, number, bottom, left, top, right);
357.     printf("%cP%c%d;16/;%s%c\\", ESC, TILDA, number, label, ESC);
358.     }

359.     /*This function returns a pointer to a string.*/

360.     #include <stdio.h>

361.     char *readin(string)

362.     char *string;
363.     {
364.     int len;
365.     char *malloc(), *ptr;
366.     len = strlen(string);
367.     ptr = malloc(len + 1);
368.     if (!ptr) {
369.         printf("memory request failed\n");
370.         exit(1);
371.     }
372.     strcpy(ptr, string);
373.     return(ptr);
374.     }

375.     /* This function prints a non-sensitive button labeling the menu options. */

376.     maketoplabel(string)

377.     char *string;
378.     {
379.     char line[80];
380.     strcpy(line, "MENU OPTIONS\\: ");
381.     strcat(line, string);
382.     makelabel(0, line, SCREENHEIGHT-LHEIGHT, HSPACE, SCREENHEIGHT,
383.         HSPACE+LLENGTH);
384.     }

385.     /* This function prints a non-sensitive button labeling the switching buttons.*/
386.     makelowerlabel(buttonnumber, page)
387.     int buttonnumber, page;
388.     {
389.     char line[80];
390.     strcpy(line, "Touch a button to see more options.");
391.     if (page == 0) makelabel(buttonnumber, line, BOTTOM+SHEIGHT+VSPACE, HSPACE,
392.         BOTTOM+SHEIGHT+VSPACE+LHEIGHT, HSPACE+LLENGTH);
393.     else printf("%c[>%d;%d;0C", ESC, buttonnumber, buttonnumber);
394.     }

395.     /* This function gets the next "word" from "string." */
396.     getword(word, string)
397.     char *word, *string;
398.     {
399.     int k = 0;
400.     while (*(string+k) == ' ' || *(string+k) == '\t') k++;
401.     while (*(string+k) != ' ' && *(string+k) != '\t' && *(string+k) != '\0') {
402.         *word++ = *(string+k);
403.     }
404.     }
405.     *word = '\0';
406.     return k; /* k points to the next character in string */
407.     }

```